A deductive database management system in Prolog

Gelsing, P.P.

Publication date:
1989

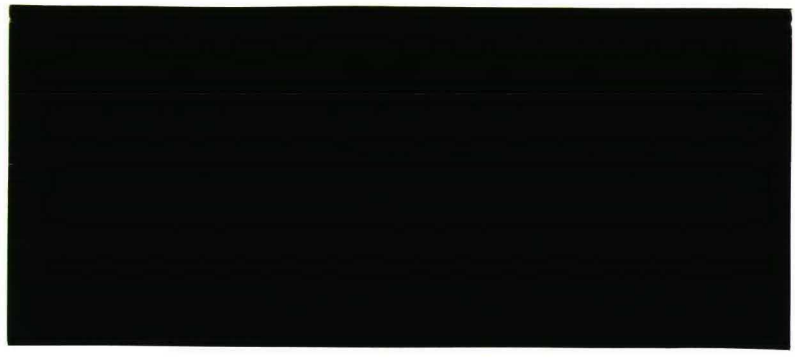
Document Version
Publisher's PDF, also known as Version of record

ITK

MEMO

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE

INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE



I.T.K. Research Memo no. 4
November 1989

A Deductive Database Management System
in Prolog

Paul P. Gelsing

Institute for Language Technology and Artificial Intelligence,
Tilburg University, The Netherlands

Table of Contents

Introduction 3

PART I Theoretical Background 4

1. An Informal Introduction 4
2. Formalizing Deductive Databases 5
3. Database Programming in Prolog 7
4. Prolog in the Database Context 11
5. Conclusions of Part I 12

PART II Implementation of a DDBMS 13

1. About KBMS5 13
2. Knowledge Representation 13
3. Data Manipulation 20
4. Data Retrieval 23

Remarks and Conclusions 29

1. Improvements to KBMS5 29

References 31

appendix 1 An Example of Query Optimization 32

appendix 2 Elaboration of Formulas in II.4.2. 33

appendix 3 Listing of the Program 36

Introduction

The aim of the project that led to this report, was to investigate the area of deductive database research. That is, the main characteristics of deductive databases should be described.
Furthermore, an implementation of some ideas stemming from the investigation should be carried out. The application language was the logic programming language Prolog. This implies that the applicability of Prolog in the database field was to be tested.

A deductive database is a generalized relational database, in which complex pieces of knowledge can be represented. Therefore, a deductive database is a semantically rich tool to describe some universe of discourse.

The core of this report consists of two parts:
- Part I explains the theoretical foundations upon which the concept of deductive database can be built. A model that formalizes this concept is developed, based on a logical reconstruction of the relational model. Furthermore, Prolog, being one of the most suitable languages for building deductive databases, is discussed.
- Part II describes the implementation of a deductive database management system called 'KBMS5'. Essential design and implementation issues like knowledge representation, data manipulation and data retrieval are described, together with the rationale that underlies them. Special attention is given to the subject of query optimization.
The final section suggests a number of improvements to KBMS5 and provides some personal comments on the project.

It should be noted that this report requires some foreknowledge from the reader. This applies especially to the theory of relational databases and the first order predicate logic. A thorough description of the relational model can be found in [Maie83]; a good introduction to predicate logic, especially in connection with logic programming, is given in [Gene87].

PART I Theoretical Background

The first part of this report sheds light upon some theoretical principles of deductive databases and describes a tool for building them.

Section 1 tries to give the reader a feeling for the topic by way of an example;
Section 2 explains the logical foundations upon which deductive databases are built;
Section 3 is devoted to Prolog, a logic programming language that can be used for building deductive database systems;
Section 4 finally, discusses the advantages and disadvantages of actually using Prolog for building them.

1. An Informal Introduction

Consider a small company with a relational database that, among other things, contains the following two tables:

`employee`

name	group
john	10
bill	10
ray	14
art	14

`group`

group	salary
10	20,000
11	30,000
14	60,000

where the `employee`-table states in what salary group the employee is in; the `group`-table records the height of salary for every group.
Now suppose that the company would like to have a list of all managers. Because it is known that every employee earning at least 40,000 is a manager, one could create a new table `manager`. For every employee it is checked whether he is in a salary group corresponding to a salary of at least 40,000, and if he is, a manager is added. Furthermore, whenever an employee is removed, we might have to remove his occurrence from the `manager`-table.

However, if the database grows, a much more convenient solution is to state the general knowledge about who is a manager in the database itself. This might look like

```
if employee(Name,Group) and group(Group,Salary) and
S ≥ 40,000 then manager(Name,Salary).
```

Such a piece of knowledge is called a **deductive law**; the table `manager` is an **implicit relation**, since its contents are not entered explicitly, but derived through a deductive law.

The former constitutes a simple example of what is called a **deductive database**:

A deductive database is a generalized relational database, consisting of explicit facts as well as general rules.[1]

A major advantage of deductive databases, as opposed to their relational counterparts, is their increased expressiveness. We can explicitly state general knowledge about a whole group of objects without having to know exactly what individuals we are talking about. Also, we can explicitly restrict the number of possible states the database can be in (integrity constraints). In conventional systems, the latter kind of knowledge is usually embedded in pieces of program that control the manipulation of data. Therefore, it suffers from the drawbacks of procedural knowledge: harder to understand and to modify.
Another important advantage stems from the use of logic for deductive databases. Logic is a formal language for expressing knowledge as well as a set of rules that dictates how new statements can be derived from old ones. This means that logic, being a language with a well understood semantics, provides a uniform framework both for data modelling and knowledge deduction.

The problems most unique to deductive databases are those stemming from the use of general rules. These rules complicate the manipulation of data, the protection of consistency and the efficient retrieval of data. Other problems resemble conventional database research topics: efficient storage of large portions of data, recovery, choosing a proper data structure etcetera.

2. Formalizing Deductive Databases

In this section the logical foundations of deductive databases are described. Details of what is shown here can be found in articles by Reiter ([Reit86]) and Gallaire, Minker and Nicolas ([Gall84]).

2.1.:Since a deductive database is defined to be relational, it is shown how the relational data model can be viewed as a special kind of first order theory.
2.2.:A formal definition of deductive databases is given;
2.3.:In this subsection, it is shown how this definition can be adapted by adopting meta-rules as to allow for a reasonably efficient implementation.

2.1. The Relational Model as a First Order Theory

We can view a logic relational database in two ways: model theoretic and proof theoretic. In the model theoretic approach, the value of a query is determined by those instances of its free variables that make the query true with respect to the interpretation, which is the set of facts in the database. An integrity constraint is said to be satisfied by the database iff this database is a model of the constraint (that is, when every possible instantiation of the free variables in the constraint makes it true).
This way of viewing databases however, has some drawbacks:

---

1 This is an informal definition that will be made more accurate later on.

- it is difficult to represent disjunctive information, for example 'paul is in salary group 10 or 14, but I don't know which of these',

- it is hard to represent null values in a way that properly reflects their meaning, and

- the relational formalism is not expressive enough to incorporate several kinds of semantic knowledge, like general facts or generalization hierarchies.

Reiter therefore advocates the proof theoretic view of databases. In this view the database is considered a **relational theory**. The interpretation for such a theory is called a **relational interpretation**, that is, an interpretation in which every constant has a 1-1 mapping with an individual in the domain. An answer to a query now is a vector of constants from the database for which the query is provable from the theory; an integrity constraint is satisfied iff it can be proven from the theory. This theory is build from standard syntactic elements and consists of the following formulae:

a) The ground atomic facts, like

`employee(john,10)`

where 'ground' means ' without variables'.

b) The Domain Closure Axiom, stating that the database individuals are the only existing ones. The DCA might look like

$\forall$ `X : X = john` $\vee$ `X = bill` $\vee$ `. .` $\vee$ `X = 60,000`

We need such an axiom because otherwise we could not, for example, prove

$\forall$ `X,Y : employee(X,Y)` $\vee$ `manager(X,Y)`

c) The Unique Name Axioms, stating that different constants denote different individuals, like These axioms are needed, since a statement like

`not = (john,art)`

is true in the interpretation but unprovable without the UNA.

d) Equality axioms concerning reflexivity, commutativity, transitivity and substitution of equal terms. These axioms formalise our intuitive meaning of equality.

e) Completion axioms for all predicates. In order to treat negated formulae correctly, we 'link together' every predicate with the set of constants that may be instances of it. This implies rules like

`if manager(X,Y) then (X = ray` $\wedge$ `Y = 14) or (X = art` $\wedge$ `Y = 14)`

Now we can prove, for example,

`not manager(bill,10)`

Reiter shows that the concept of provability in a relational theory is equivalent to the concept of truth in the relational interpretation. Thus, a formula that can be proven from the theory is true in the interpretation, and vice versa.

The improvement made by the proof theoretic view, is that the problems mentioned above, can now be solved more easily. Consider for example the representation of disjunctive information. In the model theoretic approach we have to create **n+1** interpretations to represent a formula of **n** disjunctive parts. Now the disjunction is true since in every interpretation at least one of its disjuncts is true.

A theory however, can be adapted to represent disjunctive information by simply extending one or more completion axioms. If, for example, we want to state

`employee(paul,11)` $\vee$ `employee(paul,14)`

the completion axiom for `employee` becomes

`if employee(X,Y) then (X = john` $\wedge$ `Y = 10)` $\vee$ `....`

$\vee$ `(X = paul` $\wedge$ `Y = 11)` $\vee$ `(X = paul` $\wedge$ `Y = 14).`

2.2. A Formal Definition

To build a theory for deductive databases, some adaptations to the previously described theory need to be carried out.

We should first add a new class of axioms: the general rules. These laws are formulae that are supposed to be functor-free; furthermore, their clausal form needs to be definite, that is, it may contain at most one positive literal. [1]
Another adjustment concerns the predicate completion axioms. To account for the general rules these axioms must not only refer to individuals, but also to rules, for example

```
if manager(X,Y) then (X = art ∧ Y = 14) ∨ (X = ray ∧ Y = 14) ∨
(employee(X,G) ∧ group(G,Y) ∧ Y ≥ 40,000).
```

Summarising, a deductive database **consists of:**
(1) a theory T as defined in 2.1., and adapted as described above, plus
(2) a set of integrity constraints, being a number of any closed formulae.
The database obeys these constraints if every one of them is provable from T.

2.3. Meta-rules

A straightforward implementation of a deductive database system as a theorem prover for a theory as described, is not recommendable. For one faces the problem that the number of axioms responds strongly to the addition of individuals to the database. This makes the implementation prohibitively expensive for any other than a trivial database application.
Fortunately, we can adopt some meta-rules. These meta-rules do not concern specific facts, but rather state something about how to manipulate a whole class of statements. Thus they can replace several axioms.

- The clausal form of any formula must be **range-restricted**: this means that every variable in the conclusion of a clause appears in its antecedents too. For example,

```
if employee(Name,14) then manager(Name,Age)
```

is not range restricted since Age is not defined to reach over some domain. This implies that in order to prove this formula, we need the domain closure axiom; as shown by Nicolas ([Nico79]), the DCA is not needed when all clauses are range restricted.
- A very important meta-rule is **negation-as-failure**. This rule states that ' P' can be inferred if every attempt to prove 'P' fails. Adopting negation-as-failure implies that we can replace all completion axioms. For it can be proven that under the negation-as-failure rule the answers obtained by evaluating an expression 'Q' are the same as the disjuncts in the antecedents of the completion axiom for 'Q' ([Clark78], p. 312).
- The equality axioms were only needed for use in the representation of the abolished axioms and therefore can be thrown away.

3. Database Programming in Prolog

For building a deductive database system, we need a programming language. As mentioned in section 1, a major advantage of deductive databases is the uniformity of representing as well as manipulating knowledge through logic.

---

1 Readers not familiar with logic programming terminology should refer to 3.1. first.

3.1.:Being the most important logic programming language, Prolog is discussed;
3.2.:An attractive feature of Prolog for building deductive database systems, meta-programming, is described.

3.1. Prolog

Prolog (an abbreviation of 'PROgrammation en LOGique') was first implemented by Colmerauer et al. in Marseille in 1973. Its performance, that initially was very poor, was greatly improved by Warren and Pereira on a DEC-10 computer in Edinburgh in 1977. A major impulse to the research on Prolog (and to logic programming in general) came from the announcement of the Japanese fifth-generation project in 1981. In this project, logic programming and hardware on which logic programs can run efficiently are the basic technologies.

A Prolog program is built upon a set of **Horn clauses**; a **clause** is a set of positive or negative atomic formulae (called **literals**); a Horn clause is a clause that contains at most one positive literal. Therefore, only a subset of all predicate logic formulae can be rewritten in Horn form. This translation to a Horn clause is quite straightforward, as the following simple example shows:

$\forall$ `N,G,S:[employee(N,G)` $\wedge$ `group(G,S)` $\wedge$ `S` $\geq$ `40,000` $\rightarrow$ `manager(N,S)]`

Because of the scheme '$a \rightarrow b \equiv_{not} a \vee b$' we may write

$\forall$ `N,G,S:[not (employee(N,G)` $\wedge$ `group(G,S)` $\wedge$ `S` $\geq$ `40,000)` $\vee$
`manager(N,S)]`

Then we apply '$_{not}(a \wedge b) \equiv_{not} a \vee_{not} b$', thus getting

$\forall$ `N,G,S:[not employee(N,G)` $\vee$ `not group(G,S)` $\vee$ `not S` $\geq$`40,000` $\vee$
`manager(N,S)]`

Since this formula contains only one positive literal, we can directly write down its Horn clause form:

`{not employee(N,G), not group(G,S), not S` $\geq$`40,000, manager(N,S)}.`

The Prolog notation for such a Horn clause is

`manager(N,S) :- employee(N,G), group(G,S), S` $\geq$`40,000.`

The interesting thing about such a Prolog statement is that it can be viewed in either one of two ways. Procedurally, we can describe it as "To find an answer to manager(N,S), first find an employee(N,G), then find the corresponding answer to group(G,S) and finally, check if this S is at least equal to 40,000". Declaratively however, this clause can be read as "N is the name of a manager earning a salary of S if there's an employee with name N in group G where G corresponds to a salary S of at least 40,000".
These two views make clear that Prolog is both a programming language and a language for expressing knowledge. The latter property implies that Prolog programs are easy to understand since they describe the nature of the problem rather than stating how to solve it.

Any Prolog program constitutes a theory. The task of the interpreter of the language is to show whether or not some clause (called a **goal**) is a consequence of this theory. This is done via a **refutation proof**, such a proof relies on the knowledge that if it is possible to derive a contradiction when the negation of a goal is added to the theory, this goal must be a theorem.
To make the derivation of a contradiction be carried out efficiently, Prolog uses a special kind of proof procedure, called **resolution**. This procedure derives new axioms from the theory by taking two clauses, one of which contains a positive literal and the other the negated version of a matching literal. Roughly spoken, two literals are said to **match** if their predicate names are the same and their arguments can be made equal. For example, `manager(X,14)` and `manager(art,Y)` match with **substitution** `X = art` and `Y = 14`.
In order to make clear the specific Prolog strategy, called **SLD-resolution** (for "Linear resolution with Selection function for Definite clauses"), consider the following example, taken from [Frost, p. 241].
We have a theory containing the following clauses:

```
{not K,M},  {not L,K},  {not K,not L},  {not M,L}.
```

Now we want to prove **not L and not K**. We take the following steps:

```
1) Add the clausal form of the negated goal, {L,K},
to the theory,[1]
2) Resolve {L,K} with {not K,M},          getting {L,M},
3) Resolve {L,M} with {not M,L},          getting {L},
4) Resolve {L} with {not L,K},            getting {K},
5) Resolve {K} with {not K,not L},        getting {not L},
6) Resolve {not L} with {L},          getting {}.
```

Deriving a clause with no literals means that we have detected a contradiction, so the original goal must be a consequence of the theory.

It should be noted that this proof sequence is not the only possible one. In general, especially when predicates with variable arguments are involved, there will often be a number of possible sequences. These sequences form a search tree that is traversed **depth-first**, that is, every sequence is fully developed until either a contradiction is found or no more resolution steps can be taken. In both cases Prolog **backtracks**: it traverses upwards through the tree, possibly undoing some substitutions, until a point is reached where another path may be explored.

The strategy of Prolog in proving goals is independent of the problem on hand. That is, the user states what to prove, and Prolog concerns itself with the traversal of the search tree. A drawback of this exhaustive and non-heuristic search strategy is its inefficiency. That is, Prolog may search paths that will never lead to an answer. In order to gain some control over the search process, Prolog has a built-in control primitive called 'cut' (written '!').

As an example of the use of the cut, consider the following program:

```
minimum(X,Y,X) :-       % X is the minimum of X and Y if
      X ≤ Y, !.         % X ≤ Y;
minimum(X,Y,Y) :-       % Y is the minimum if
      X > Y.            % X > Y.
```

We know that **X** ≤ **Y**and **X** >**Y** exclude each other, so the program is deterministic. The advantage of using the cut then is that, after one of the conditions is seen to be true, the other clause will not be encountered on backtracking.

Informally spoken, the cut prunes the search tree in such a way that all paths, traversed before the cut was encountered, are not evaluated again on backtracking.

3.2. Meta-level Programming in Prolog

Meta-knowledge could be described as 'knowledge about knowledge': we are not interested in the contents of the piece of data, but we record properties of this piece. For example, if we have **man(john)** and **man(art)**, a piece of meta-knowledge might be "**man(john)** and **man(art)** are instances of the set **Men**"

In predicate logic, this knowledge is represented by second-order predicates. While the one-place predicate **dog** can be viewed as denoting the set of all dogs, the second-order one-place predicate **mammal** denotes the set of all mammals. In this set, **mammal(dog)** is an element. Of course,this process can be repeated, getting third-order predicates like **living(mammal)** etcetera.

One of the attractive features of Prolog is its uniformity of representation; that is, every clause is both a piece of data and a piece of program. This makes meta-programming, writing programs

---

1 {L,K} is *not* a Horn Clause!

that treat other programs as data, a natural activity. One can, for example write a Prolog interpreter in Prolog (and a very short one too: `interpret(A) :- A.` !).
There are some built-in meta-predicates in Prolog. Normally, all solutions to a goal have to be generated one by one; every time a new one is found, the old one is not accessible any more. The meta-predicates, like `bagof`, obtain a list of all solutions.
In building knowledge base systems, two advantages of meta-programming are important:

1) One gets control over the computational process. Noteworthy application of this control is influencing the selection function of Prolog. Normally, if we query a database with `:- a,b,c`, we are sure that these subgoals will be evaluated left-to-right. However, we can consider `a,b` and `c` to be arguments of a predicate, say `q`, so that a query becomes `:- q(a,b,c)`. Now we can choose the order in which the subgoals are to be evaluated ourselves.[1]
Another interesting application is the recording of information on the deduction process. This is especially useful in building expert system shells, where the ability of the system to explain its own behaviour is an essential feature.
In deductive database systems finally, it can be shown to the user what pieces of information are deducible from explicitly given ones.

2) One can retain information on information. Uncertainty reasoning in expert systems, for example, might need information on the probability of rules.
In a database, meta-information can appear in various forms:
- describing properties of one piece of information, e.g.

```
tuple(inserted_at(01-24-89),man(john)).
```

which denotes the fact that the information `man(john)` was recorded in the database at `January 24th 1989`.
- describing properties of a group of data, e.g.

```
number_of_tuples(man,10).
```

stating that the relation `man` contains 10 tuples.
- imposing conditions on the contents of (part of) the database, like

```
constraint((Address = Address1 :-
man(Name,Adress), man(Name,Adress1)).
```

describing the functional dependency of `Address` on `Name`. There are several reasons for representing such a constraint at meta-level:

* the predicate = cannot be used as head of a clause;so we would need to write something like `Address equal Address1` and add another clause `X equal X`.

* we can not check the constraint by posing it as a query, for its negation is not a Horn clause. This means that we need to write the constraint like

```
inconsistent :- man(N,A), man(N,A1), not A = A1.
```

However, this leads to inefficiency, for

* we cannot record knowledge on a constraint, so all constraints need to be checked in order to detect possible inconsistency.

Of course, a price has to be paid for these improvements. An obvious drawback is the diminished efficiency when simulating Prolog processes in the language itself. The size of the loss is dependent on how detailed our meta-interpreter is. On one side of the spectrum, `interpret(A) :- A`, the efficiency loss is minimal, and so is the scope for applying such an interpreter (for we do nothing with `A` before letting Prolog evaluate it). On the other side, all processes are simulated, including choice of clauses, backtracking, unification etcetera. It is not easy to think of an application that benefits from so detailed an interpreter.

---

1 Currently there are Prolog implementations using either a more sophisticated selection function themselves, or postponing certain kinds of subgoals, e.g. negative non-ground ones.

The meta-interpreting on clause level, that is, manipulating clauses but leaving everything needed for evaluating a single literal to Prolog, seems to be the most suitable for knowledge base manipulation. The objective then is, to regain the lost evaluation time by reordering a set of subgoals, so that their joint evaluation will become faster.
A second disadvantage is the extra care that has to be taken when manipulating programs containing variables in various clauses. For example, serious problems can arise concerning renaming of variables. More on this topic is found in the section about query optimization.

4. Prolog in the Database Context

This section investigates the suitability of Prolog as a tool for building deductive databases.

4.1.:It is described what kind of knowledge can be represented;
4.2.:Some major drawbacks of using Prolog are noted.

4.1. Pros ...

Any database is a model of a small, finite subset of some world. One might say that a database is a good model if a natural correspondence exists between constructs in the world and in the database.
The question at this point is: does a Prolog deductive database provide such a correspondence, in other words, what can we express in Horn clause logic?

- Objects can be described by a predicate name and a number of arguments. For example,

```
person(wiley,london,68).
```

might denote the individual named `wiley` with residence `london` and age `68`.
- Groups of objects, are described in the same way, for example

```
person(Name,Residence,68).
```

denoting a class of persons having age `68` (this is called **classification:** every individual is associated with some type).
- Relations between objects or groups of objects can either be described by a single literal, like

```
family(father(john),mother(mary),children([art,bill])).
```

or by a clause representing a deductive law, like

```
mammal(Animal) :- dog(Animal), alive(Animal).
```

- Restrictions on relationships are described by clauses with a non-empty body. In database terms they are called integrity constraints [1]. An example is

```
:- mammal(Animal), bird(Animal).
```

- Restrictions on possible states of the world we describe, called 'transition laws', need a special treatment. They can nevertheless be represented if we define for every manipulation operation a new relation that consists of pairs of states, denoting what transitions may take place. [2]

4.2. ... and Cons

---

1 Strictly spoken these concepts are not equivalent since restrictions are imposed on the conceptual model.

2 See for a clarification of this point [Nico78], pp. 336-

As noted earlier, logic provides a well-defined framework for representing and reasoning with knowledge. Using Prolog, an additional advantage is that we have a programming language at our disposal. This implies that, as opposed to conventional systems, a deductive database system may entirely consist of Prolog programs. That is, Prolog serves as an expressive data definition language, a query language, a database management system and a language in which application programs can be written.
Although all this is quite advantageous, a few marginal notes are in order here:

- Prolog's expressive power, although seemingly exceeding 'classical' database description languages, is limited in its expressive power. The restriction to Horn clauses makes it, for example, impossible to state disjunctive facts. It should be noted however, that research is conducted on possibilities to make Prolog allow for more predicate logic formulae (see e.g. [Lloy83b]).

- Prolog lacks efficiency. A database management system should be able to handle large portions of data efficiently. Prolog offers few possibilities for arranging data on secondary storage media and for input/output operations. Possible solutions to these problems are the interfacing of a Prolog deductive system with a conventional DBMS that does the actual storage and retrieving.

Furthermore, Prolog provides rather poor control facilities, that is, it is difficult to interfere with the course of a program.

The conclusion to be drawn at this point is that Prolog is an important tool for designing and building semantically richer databases; as a tool for building practically usable management systems, it is at present quite inadequate.

5. Conclusions of Part I

In this part of the report, the foundations of deductive databases were elaborated; furthermore, the logic programming language Prolog was described. The most important conclusions to be drawn are:

1) Deductive databases are an interesting extension to conventional relational databases. This is true for several reasons.

- First, more complex knowledge can be represented through general rules. Of course, one could argue that in most relational systems the view is an equally powerful concept. However, a general rule is advantageous in that it is formulated as part of the conceptual model and therefore is a higher level concept. Besides, rules can be recursive and finally, rules provide a uniform representation formalism for expressing derived relations and integrity constraints.

- Another important characteristic of deductive databases is the use of first-order predicate logic for expressing and manipulating knowledge. Some advantages of logic are its precision and unambiguousness of meaning, its transparentness and its declarativeness.

2) The declarative character of a logic data model implies that it can be stored and handled by various kinds of languages. The focus in this report is on a programming language that is based on logic itself, namely Prolog.

- The nice thing about Prolog is that it is mainly declarative, so that adesigner can concentrate on formulating the data model rather than inventing ways to handle it.

- Prolog has at present some serious drawbacks, of which its restriction to a subset of predicate logic and its inefficiency are the most important.

PART II Implementation of a DDBMS

The second part of this report is devoted to the description of KBMS5, a prototypical deductive database management system that was implemented at Tilburg University.

Section 1 provides a short introduction to the program;
Section 2 describes how knowledge is represented in the deductive database and the reasons for representing it this way;
Section 3 describes how KBMS5 fulfills the task of manipulating data and preserving data integrity in the presence of general rules;
Section 4 finally, discusses data retrieval; this section is largely devoted to the description of attempts to make the Prolog evaluation of queries as efficient as possible.

1. About KBMS5

KBMS5 is a (prototypical) deductive database management system written in Prolog. It operates on a user-defined database, also expressed in Prolog. This database contains a set of facts, a set of deductive laws and a set of integrity constraints. These constraints restrict the number of possible states the database can be in.

The management system consists of two modules:

1) A query module. This program handles all user requests for data. It checks queries for errors, performs a reordering of the subgoals in a query if this will make it to be evaluated more efficiently, and retrieves the requested data.

2) A knowledge manipulation module. This part manipulates the knowledge through insertions and deletions, while maintaining database consistency as defined by the integrity constraints.

'KBMS5'™ abbreviates 'Knowledge Base Management System 5'. It is called this way since representing more complex pieces of data leads to a knowledge base rather than to a database. Theoretically it is desirable to retain a distinction between knowledge, data and information; for the sake of convenience however, this report relies on notions that are intuitively clear to someone familiar with database terminology.

2. Knowledge Representation

In this section we discuss the way in which data are represented in the database.

2.1.:The format of facts, deductive laws and integrity constraints is described;
2.2.:It is discussed how explicitly entered information and derived information are treated coherently;

2.3.:In this subsection, a method for determining which attributes of a derived relations have a uniqueness property is developed.

2.1. Format of Data

The simplest, and in fact most natural way, to represent facts in a database would be to use the name of a relation as a predicate. Thus, `man(john)` and `man(art)` might be two tuples from the relation `man`. But, as explained in part I, lifting these tuples to the meta-level allows us to record knowledge on the tuples that could not be represented otherwise (well, not without doing violence to a clear declarative meaning of Prolog atoms anyway). For similar reasons, deductive laws and integrity constraints are also represented at meta-level.

Representing Facts
We define the format of a tuple to be

```
tuple(Status,Occurrences,Relation(Tuple)).
```

- The argument `Status` either states that a tuple is accepted as being a fact and can be used while querying (`Status` = 'accept'), or that an occurrence of a tuple is proposed for insertion (`insert` + N) or deletion (`delete` - N). N is a natural number, denoting the proposed increase or decrease of the `Occurrences` argument. The reasons for formatting `Status` this way will become clear later on.

- `Occurrences` is a natural number that denotes the number of times the tuple has been deduced (in this case, an assertion by the user of the tuple is called a deduction too). Important is, that the value of the `Occurrences` argument does not say anything about the world we are representing. That is, if `Occurrences` is 3, we know that the fact has been deduced thrice; but of course this does not mean that we are representing three facts. `Occurrences` is merely a piece of control information to make sure, for example, that when a tuple has been deduced from two sources and only one of those sources is deleted, the tuple itself is not. In that case, only the value of `Occurrences` is decremented from two to one.

Representing Deductive Laws
A deductive law is expressed as

```
dlaw((Head :- Body)),
```

where `Body` is a normal Prolog clause body and `Head` is composed of a relation name as predicate and a number of arguments that, if variable, appear in `Body` (range restrictedness).
This meta-level format prevents that facts, being both deducible through deductive laws and through facts, are retrieved twice on queries. Furthermore, the program now can easily discriminate between deductive laws and integrity constraints (through which the burden of deciding on the nature of a clause that says something about the database, now is borne by the designer).

Representing Integrity Constraints
An integrity constraint is written as

```
ic((Head :- Body),Type).
```

- The first argument is a clause that is not necessarily a legal Prolog clause. `Body` must be a normal Prolog clause body; `Head` however, can be any Prolog literal, like `true`, `false`, `A = B` etcetera.
Interesting application of the latter, though quite dangerous, is to use the `ic`-clause both for expressing the integrity constraint and for inducing automatic recovery.
Suppose for example, that we have a database containing a relation `man(Name,Address)`. If we want to state the integrity constraint that no two men can have the same name, we could write

```
ic((Y = Y1 :- man(X,Y), man(X,Y1))).        % (type omitted)
```

Whenever the body of the constraint is `true`, the program checks whether the head is `true` too. If it is not, the constraint is violated and the program suggests a solution that restores

consistency. In this case, one of the two men that together constitute the violation should be removed from the database.
We might however, circumvent the recovery mechanism by writing a head that induces the recovery itself and thus will always be true. This might look like

```
ic((recover(man(X,Y),man(X,Z)) :-
        man(X,Y),man(X,Z),Y ≠ Z)).     % (type omitted)
recover(X,Y) :-
        choose(X,Y,Choice),
        delete(Choice).
```

After the insertion of a `man` tuple, the checking program first tries to fulfil the body of the constraint. If this can be done, we have a violation. Next it will try to fulfil the head of the constraint. This head makes the user choose between two men to remove from the database. Since the call to `recover` will always succeed (there are two men violating the constraint), the head is made true, so the constraint will be considered not violated.
Though this method, if used carefully, works correctly, it should better not be used, for it muddles the declarativity of the constraint, it becomes part of the management program (as the database should be strictly separated from it), and it interferes with the recovering strategy of KBMS5 itself.

- The argument `Type` acts as an aid for the program in determining how a possible inconsistency might be solved. On design time, the designer should write down, what kind of integrity constraint he is entering. Types are
* type1a. The constraint looks like
Relation1 :- Relation2(,Conditions)

```
        (e.g. 'team(1,X,Y) :-
                team(2,X,Y),Y > 3.'),
```

* type1b. Format **Relation :- Relations(,Conditions)**

```
   (e.g. 'team(1,X,Y) :-
                team(2,X1,Y1),team(3,X2,Y2),X2 > X1,Y1 < 4.'),
```

* type2a. Format **Condition :- Relation(,Conditions)**

```
   (e.g. 'X > 4 :-
                team(X,Y,Z),Y = 1.'),
```

* type2b. Format **Condition :- Relations(,Conditions)**

```
   (e.g. 'X > Y :-
                team(X,Y,Z),team(1,Y,Z1),Z1 = 5.'),
```

Representing Knowledge on Relations
The only predicate left to discuss now, is `relation`, that records knowledge on all current relations. Its successive arguments are `Name` (name of the relation), `Attributes` (a list of the names of the attributes), `Primary` (the rank of the attributes that are primary, also in list notation) and `Number` (the current number of tuples in the relation; at design time usually 0). An example is

```
   relation(man,[surname,first_name,address],[1,2],0).
```

stating that a man's name is unique.
The primary key arguments might be hard to determine in case of derived relations; see on this topic section 2.3.

Law or Constraint?
A problem for the designer of a database is that any clause with a non-ground head that refers to some relation, can either be viewed as a deductive law or as an integrity constraint. A general rule of thumb for deciding on this problem is that the purpose of a deductive law is to derive new information. So,

```
manager(N,S) :-
```

```
        employee(N,G), group(G,S), S ≥ 40,000.
```
is a deductive law, while

```
G = G1 :-

        employee(N,G), employee(N,G1).        % (pseudo Prolog)
```
(the functional dependency of `Group` upon `Name`), must be seen as a constraint. Likewise, a purely negative assertion as

```
:- employee(paul,10)
```
(there is no employee named `paul` in group `10`), will never produce new knowledge when using the negation-as-failure rule.
The last two clauses are not allowed in Prolog. Nevertheless, if we want to use them, we might introduce a new predicate called `inconsistent`; a call to this predicate succeeds when its body can be made true, that is, when the constraint is violated. The dependency statement would then become

```
inconsistent :-

        employee(N,G), employee(N,G1), not G = G1.
```
This formulation however, is a less concise reflection of the original logic formula. For this reason (and for others that will become clear later on), KBMS5 uses meta-predicates.

2.2. Implicit and Explicit Knowledge

It is noteworthy, that two important decisions have been made, concerning the recording of facts and general knowledge. Taken together, these decisions could be stated as "(1) All implicit knowledge is represented explicitly, (2) not necessarily vice versa".

(1) Explicating Implicit Data
We treat deductive laws as so-called **generation rules**. This means, that whenever some tuple is entered, every tuple that can be derived from it, using all applicable deductive laws, is generated and inserted.
In [Nico78b] this method is discussed, together with its alternative, called **derivation rules**. In that case implicit knowledge is not made explicit during data-entry but during query evaluation time. The advantages for using deductive laws as generation rules are twofold:

a) Queries are evaluated against explicit relations. This implies that more on a relation is known before it is queried, e.g. cardinality. This information can be used to improve query performance. Furthermore, since no deduction is needed, retrieval will be more efficient. Of course this is a choice of trading off data entry performance against query performance.

b) Consistency is more easily checked, since all derivable tuples are inserted and thus submitted to the uniform consistency check that follows any knowledge manipulation. The deduction process seems to make generation inferior to derivation. However, in the case of derivation rules, we have to find a way in which we can check whether implicitly entered information violates any constraint. This implies that the performance of using derivation will not be much better.

The main disadvantage is of course the obvious redundancy in storing knowledge, a disadvantage that might be prohibitive when building large databases.
Nevertheless, it seems that the advantages just mentioned, warrant the choice. This is especially true when the database contains a growing number of facts. Good query performance then becomes an important objective. Data entry on the other hand, will not be influenced by a the size of the database, so it seems wise to relieve query evaluation of as much work as possible.

(2) Local Observations
Derived relations can contain tuples not implied by their deductive laws. This is very useful, since it might very well be that the user is aware of some derived tuple (called a local observation), without having inserted its antecedents. A good example of this is supplied by the following extensions:

```
father              grandfather
```

```
john  al          john  kim
al    kim         john  mary
-------------------------------------------------
grandfather(Oldest,Youngest) :-
        father(Oldest,Middle), father(Middle,Youngest).
```

In this case, the tuple `grandfather(john,mary)` is a local observation. We have to allow this tuple, for perhaps the tuples `father(X,mary)` and `father(john,X)` have not been inserted, or the father of `mary` is not known, or this father is dead and therefore not part of the database.

2.3. Determining the Primary Key

As will be elaborated in 4.2., the query evaluation strategy of KBMS5 is based on heuristic rules. These rules allow for an estimation of the cost of a specific ordering of the subqueries.
One of these rules uses the fact that the evaluation of a goal in which all key arguments are instantiated can return at most one answer.
The point is however, that it can be hard to determine which arguments of a derived relation form its key. That is, given the primary key arguments of the antecedents of a law, what are the primary key arguments of the consequent?

Since a derived relational clause is assumed to be range restricted, the properties of the set of tuples it represents are completely determined by its antecedents. This allows for a parallel with relational algebra: the consequent of a deductive law can be viewed as the relation that results from a number of algebraic operations on the extensions of the relations represented by its antecedents. The benefits of attaching such a 'computational meaning' ([Ullm85], Ch. 3) to Prolog clauses , is that we can apply results from a well-known model.

Prolog Clauses and Relational Algebra
Memorizing, the relational algebra consists of three operators that manipulate relations: selection (σ), projection (π) and join (ψ). The following examples will make the equivalence with Prolog clauses clear.

Prolog clauses[1]	Relational equivalent
`A(X,Y) :- B(X,Y), X > 4.`	`A = σ[x>4](B)`
`A(X,Y) :- B(X), C(Y).`	`A = B ψ C`
`A(X,Y) :- B(X,Y,Z).`	`A = π[x,y](B)`

In general, the solutions to a call to the head of a derived relational clause can be described in relational terms as

Set of solutions = π[arguments of head](σ[conditions in body](antecedent 1 ψ antecedent 2 ψ .... ψ antecedent n)).

1 Derived relations based on several laws are not considered. They need a special treatment concerning renaming of variables; see also [Ullm85], Ch. 3.

As a cosmetic improvement, any constant in one of the antecedents should be replaced by a condition with a new variable, for example `B(X,3)` becomes `B(X,Z), Z = 3`. Now suppose we have the clause

```
A(X) :-
        B(X,Z), C(Z,3), Z > 4.
```

Then we can determine the set of solutions in relational terms as follows:

```
1) A(X) :-
        B(X,Z), C(Z,Y), Z > 4, Y = 3.
2) Take the natural join of B and C
3) Select the tuples in which the conditions hold
4) Project X on those tuples
```

So the set of solutions = π [x] (σ[z4,c=3] (B ψC)).

Preserving Key Properties

The next step is to answer the question whether the uniqueness of key arguments is preserved when an relational operator is applied. This can be done by means of inheritance rules (`Pk(R)` denotes the set of key arguments of relation **R**; $\leq$ means 'is subset of').

(1) Selection:

`A := σ[..]B → Pk(A) = Pk(B).`

Meaning: If relation **A** is obtained by selecting specified tuples from relation **B**, then **A** will inherit the key arguments of **B**.

The proof of this statement is fairly trivial: if a set contains no duplicate elements, then there is no subset of it that does.

(2) Projection:

`A := π[S]B & (Pk(B) ≤ S) → Pk(A) = Pk(B).`

Meaning: If **A** is a projection of **B** on all of **B's** key arguments (and possibly others), then **A** inherits **B**'s primary key.

From the definition of primary key we know that the combination of attributes that constitute it is unique. So any combination of these arguments and a (possibly empty) set of nonkey ones will be unique too.

(3) Natural join:

This operation is more complex since several kinds of join can be distinguished. In the following rules the set of attributes over which the join is taken is denoted by Ψ.

a) `A := B ψ C ∧`

`[(Pk(B) = Ψ ∧ Pk(C) = Ψ) ∨`

`(Pk(B) = Ψ ∧ (Pk(C) ∩ Ψ ≠ {}) ∧ Pk(C) ≠ Ψ ) ∨`

`((Pk(B) ∩ Ψ = {}) ∧ (Pk(C) ∩ Ψ = {}))]`

`→`

`Pk(A) = Pk(B) ∪ Pk(C).`

Consider an example of the third disjunctive part between '[' and ']':

`B(a1,a2,a3) ψ C(a1,a2,a4) := A(a1,a2,a3,a4).`

To prove that **{a3,a4}** constitute a primary key, we must prove two properties of them:

1) Uniqueness, that is, there is at most one combination of any two values for **a3** and **a4**, say **v1** and **v2**.

Proof: suppose **{a3,a4}** is not unique. Then there is in **A** another couple **{v1,v2}**. This would mean, that B or C contains another couple of the same values for **a1** and **a2**. However, such a

couple cannot have the same **a3** or **a4** value, since these attributes are unique. So a join would not yield the same **{a3,a4}** combination.
2) Minimality, that is, no key attribute can be discarded without destroying the uniqueness property. Proof: **a4** cannot be unique on its own in **A**, since the combination **{a1,a2}** determined by a value of **a4** can appear more than once in **B** (for they are not key attributes). The same reasoning goes for **a3**.

b) $A := B \psi C \wedge$

$$[(Pk(B) = \Psi \wedge Pk(C) \cap \Psi = \{\}) \vee$$
$$(Pk(B) \cap \Psi \neq \{\} \wedge Pk(B) \neq \Psi \wedge Pk(C) \cap \Psi = \{\})]$$
$$\rightarrow$$
$$Pk(A) = Pk(C).$$

For a proof of the first disjunction, consider:

$$B(\underline{a1,a2},a3) \ \psi \ C(a1,a2,\underline{a4,a5}) := A(a1,a2,a3,\underline{a4,a5}).$$

{a4,a5} is unique and minimal since this combination does not participate in the join; it is simply copied to A and therefore retains key properties.

c) $A := B \psi C \wedge$

$$(Pk(B) \cap \Psi \neq \{\} \wedge Pk(B) \neq \{\}) \wedge$$
$$(Pk(C) \cap \Psi \neq \{\} \wedge Pk(C) \neq \{\})$$
$$\rightarrow$$
$$Pk(A) = Pk(B).$$

Example:

$$B(\underline{a1},a2,a3) \ \psi \ C(a1,\underline{a2},a4) := A(\underline{a1},a2,a3,a4).$$

The proof of the rules not discussed here is done in a similar way and is therefore omitted.

Key Inheritance in Prolog Clauses

Now we can return to the question of inheritance in Prolog clauses. To determine the key arguments of a derived relation first rewrite the Prolog clause to a corresponding algebraic formula; next, use the inheritance rules. Consider the following example:

```
D(X,Y) :- A(X,V,W), B(Y,X), C(Y,4,V).
```

Pk(A) = {X}, Pk(B) = {Y,X} and Pk(C) = {Y}.

- Rewrite the law to

```
D(X,Y) :-
        A(X,V,W), B(Y,X), C(Y,Z,V), Z = 4,
```

- this is equivalent to

$\pi[x,y](\sigma[Z=4](A \ \psi \ B \ \psi \ C))$.

- H1 = A ψ B; Pk(H1) = {X,Y} (rule 3a),
- H2 = H1 ψ C; Pk(H1) = {X,Y} (rule 3a),
- H3 = $\sigma[Z=4](H2)$; Pk(H3) = {X,Y} (rule 1),
- D = $\pi[x,y](H3)$; Pk(D) = {X,Y} (rule 2).

It should be stressed that all the former assumes the clauses to be range restricted. Otherwise it would not be possible to use projection; besides, a not range restricted clause would have no natural counterpart in the modelling process. For it makes no sense to describe an entity in the real world only partially through other entities and leave its other characteristics unspecified.

3. Data Manipulation

Adding and removing tuples in a deductive database is a quite complex process, due to the existence of derived relations.
In this section, the way in which KBMS5 fulfills this task is discussed.

3.1.:The deductive process that takes place when database contents are altered, is described;
3.2.:The retaining of data integrity is discussed.

3.1. Updating and Deduction

In deductive databases, the scope of an update is not restricted to a single relation. It can alter the contents of other relations as well.

Update Operations
We can distinguish several kinds of operations, depending on whether we manipulate explicit or implicit relations.

(1) Inserting or deleting a tuple that is part of a relation not appearing in any deductive law. In this case, only database consistency after the update has to be checked.

(2) Inserting a consequent tuple (to be precise, "an instance of a relation that appears as head of one or more deductive laws"). This is a local observation, so the only thing to do (apart from the consistency checks) is checking whether it was deduced by one of its laws before. In that case, the tuple is not inserted again, but its occurrence is marked as being deduced more than once.

(3) Inserting an antecedent tuple. Now we need to check what tuples are derivable from the inserted one. For every derived tuple that did not exist before, an occurrence is added to the database and the insertion process is repeated.

(4) Deleting an antecedent tuple. In this case, we should find all tuples that can be derived no longer from the updated database. This process is analogous to that described under (3).

(5) Deleting a consequent tuple. Now we must find all tuples that accounted for the deduction of the deleted one, and remove just enough of these, so that the deduction is made impossible. Since often this yields various possible deletions, this case is the most complex one.

Deleting a Consequent
In order to understand the problems arising when handling the deletion of some consequent tuple, consider again the example given in the introduction.

```
employee          group             manager
_____________     _________________ _____________
john   10          10     20,000     ray    60,000
bill   10          11     30,000     art    60,000
ray    14          14     60,000
art    14
_____________________________
manager(Name,Salary) :-
```

```
employee(Name,Group), group(Group,Salary), Salary ≥ 40,000.
```

If next we obtain the information that `ray` is no longer a manager, we could delete either the knowledge that `ray` is an employee or the knowledge that the salary in group `14` is `60,000`. Choosing the latter solution however, would probably not mirror the intention of the update, since it will make the fact that `art` is a manager not derivable any longer (side-effects).
So in this case, deleting `employee(ray,14)` would be appropriate. The point to stress now is, that this choice is purely a matter of semantics (that is, dependent on the interpretation of the relations) and therefore can not be made by the program. The user should therefore play an active role in determining the antecedents to be deleted. This will also, in general, solve the second problem glanced upon, the unwanted side-effects. For, if the user chooses to delete `group(14,60,000)` after all, we may safely assume that he wishes to remove all managers from group `14` (even though he then exploits a strange way of doing so).

The method used in KBMS5, is composed of two independent processes:

1) **Backward chaining.** The user chooses an instance of a relation from every deductive law that has the initial tuple as its consequent. This process is repeated for the chosen tuples until all tuples to be deleted are terminal (that is, they are not derivable).

2) **Forward chaining**. Every tuple obtained in step 1 is deleted, together with all its derivable knowledge.
It is not too hard to automate part of the backward chaining process, so that the program computes a list of terminal tuples with minimal side-effects when deleted. This would be, however, a syntactical improvement that hardly supports the semantic decision process of the user.

Occurrence Counting
As mentioned earlier, in KBMS5 every tuple bears carries a piece of meta-knowledge with it, recording the number of times the tuple has been deduced. This is a very useful aid in the process of deleting an antecedent tuple; for if we did not record it, we would have to determine for every tuple derivable from the initial one, if it can be derived another way. Only if no other derivations were found, this consequent tuple could be removed. When using a counter, we only have to check whether the counter of the derived tuple has become 1. If not, we do not remove the tuple, but simply decrement its counter. In that case, we will not carry on the deduction process with this tuple.
Of course, the counters might be incremented on inserting tuples. For an initial tuple, we check whether it occurs already. If it does, its counter is incremented by 1; if not, we insert the tuple with counter value 1, and repeat the whole process for all derivable tuples.

3.2. Protecting Consistency

There are two important issues to be addressed concerning the consistency of knowledge. The first one is: how can it be checked efficiently whether one or more integrity constraints are violated; the second one is: what should be done to restore integrity when such a violation occurs.

Checking Consistency
The simplest, and almost inevitably least efficient way to check database integrity, is taking the first constraint, trying to find an instantiation that falsifies it and, if no such falsification can be found, taking the next constraint and so on until everything is proved consistent.
There are at least two major improvements to this method:

1) Consider only those tuples that might cause a violation,
2) Consider only those constraints in which some literal matches some tuple found in 1).

Candidate tuples are those being proposed for insertion or deletion. The number of candidates is usually only a small percentage of the total number of tuples. So, together with the fact that fewer constraints are considered, this method significantly gains efficiency as compared to the first one.

In KBMS5, the checking strategy is a slight refinement of this method. It uses the knowledge that a deletion cannot invalidate a constraint of type **Condition :- Relation(,Conditions)** (type2a), since a deletion could only make the body true if 'Relation' is negated. This would however, lead to evaluation of a not fully instantiated negated goal.
As for the second issue, restoring integrity after a constraint violation, one has to decide whether a database should be consistent at any point in time. Deciding it has to implies that database consistency will be checked immediately after the insertion or deletion of knowledge. Various steps might then be taken.
However, maintaining consistency in such a strict way does not seem to be an acceptable strategy. If, for example, we want to add both tuples of a foreign key relationship, the database is inconsistent after inserting the referring tuple. What we want though, is integrity to be checked only after the referred-to tuple has been inserted too.

KBMS5 therefore allows the database to be in a temporary state of inconsistency. Only after issuing a commit command constraint checking is performed. As long as no committing has taken place, the changes are assumed not to be made, so querying is done on database contents as they were directly after the last commit. It can be argued that conceptually, this is the most appropriate method, since otherwise we might obtain knowledge that is not a correct reflection of the world as we perceive it.

Restoring Consistency
The next question to be addressed is: how should the system react when committing is impossible? One extreme is to roll back all operations automatically, that is, to restore database contents to the state it was in before any change was made. This would however not be terribly apt, especially not for deductive databases. The reason is that manipulating one tuple might induce the updating of several others, all of which might violate constraints themselves. Since these effects can be hard to foresee, the user should have the possibility of introducing additional changes after committing failed.
The other, seemingly most intelligent extreme, is to generate additional operations in order to restore integrity automatically. Problems in this case stem from the complexity of these operations in the presence of deductive laws, and from the fact that more often than not integrity might be restored in several ways. The latter problem is analogous to the deleting of a consequent information. Consider, for example, the `employee-group-manager` database from the previous section and suppose the `manager`-law is interpreted as an integrity constraint. When we insert `employee(paul,14)`, we violate this constraint; now there are two possible revalidations: inserting `manager(paul,60,000)` and deleting `group(14,60,000)`. Again, the choice is a semantic one and can only be made by the user (though one might record for every constraint which one of several possible revalidations should be chosen).

KBMS5 takes its place somewhere in the middle of the spectrum. Whenever a commit fails, the operations are not denied, but the user is supplied with the reasons for the appearance of inconsistency, together with a set of possible revalidation solutions. This set is not guaranteed to liquidate all inconsistency, for issuing it might affect other tuples.

Including the type of a constraint as meta-information facilitates the determination of ways in which integrity can be restored:

- type1a, e.g.

```
team(1,Y,Z) :-
        team(2,Y,Z).
```

If the constraint is invalidated by (an instantiation) of the antecedent or deleting (an instantiation) of the consequent, the suggested solution is to insert a matching instantiation of the consequent or to delete a matching instantiation of the antecedent respectively.

- type1b, e.g.

```
team(1,Y,Z) :-
        team(2,Y,Z), team(3,Y1,Z1).
```

In case of violation through deletion of the consequent, a matching tuple to one of the antecedents should be deleted; in case of violation through insertion, a tuple matching to the consequent should be inserted.

- type2a, e.g.

```
X > Y :-
        team(X,Y,Z).
```

When a constraint of this type is violated, no revalidation is possible (since it can only be violated by inserting the antecedent; then there is no way to make the consequent true). All previous transactions are rolled back.

- type2b, e.g.

```
Y > Z :-
        team(1,Y,Z1), team(2,Y1,Z).
```

If this kind of integrity constraint is violated, through insertion of an antecedent tuple, it can be revalidated by deleting a matching tuple to another antecedent. Obviously, this constraint cannot be violated through a deletion (assuming the body does not contain any negated subgoals).

When the database is found to be consistent, all proposed changes are committed. If it is not, the user can issue the rollback command or introduce additional changes to the database.

4. Data Retrieval

Central to every DBMS is the ability to retrieve requested data efficiently. For large database applications this is not a trivial matter.
We can roughly divide the factors influencing the cost of retrieving some set of data into two groups.

4.1.:This subsection briefly glances at physical storage, that is, how can database accesses be facilitated;
4.2.:The main part of section 4 is devoted to query optimization, that is, how can an arbitrary query be transformed into a query that can be evaluated more efficiently.

4.1. Physical Storage

In conventional systems, the process of locating an item of data and presenting it to the user, is highly layered. The DBMS receives a request for data and decides what records are needed. The 'file manager' then decides what page (the unit of information transferred in a single disk access) contains the desired record and finally, the 'disk manager' determines the physical location of that page and issues the I/O-operation.
Since access to main storage is much faster than disk access, minimizing the number of I/O-operations is an overriding performance objective. The aim of designing storage structures therefore is to arrange data on secondary storage so that a required piece of data can be located in as few I/O's as possible. Techniques for doing so include indexing, hashing, pointer chains and compression (see for details any introductory book on databases).

When implementing a database system in Prolog, one faces the fact that very little can be done with respect to physical data storage.[1] This is largely due to the lack of procedural control: we describe what we want to store or retrieve, rather than how to do it. This problem can be solved by either interfacing the Prolog system with a conventional DBMS that does all the retrieving, or by implementing a suitable file structure within Prolog itself, both of which are very complex.
KBMS5 operates on a database system that is entirely kept in main memory. The efficiency problem then becomes: how can a user's request be transformed in such a way that the Prolog deduction system will come up quickly with the desired answers? This problem leads us into the area of query optimization.

4.2. Query Optimization

A database query in Prolog is a conjunction of subgoals. An answer to such a query consists of those bindings for the variables occurring in the query that make all of the conjuncts true. Of course, logically the order of the subgoals has no meaning. For the Prolog resolution strategy however, a well-considered reordering of subgoals can make a lot of computational difference. Intuitively, the intention is to reduce the number of times the deeper levels of the search tree are encountered. This will be formalized now. [2]

Definitions
A **query** is a list of subgoals. A **subgoal** can be
- **relational**. The subgoal is then of the form **R(X1 .. Xn)** where every **Xi** is either a variable or a constant and **R** refers to a set of ground clauses in the database, e.g. `employee(ray,14).`
- **conditional**. The subgoal is an expression that constrains the domain from which the value of some variable can be taken, e.g. `G > 10.`
- **implicit relational**. Instead of referring to a ground clause, the subgoal now refers to a program clause of the form **R(X1 .. Xn) :- Q** where **Q** is a query. This clause is assumed to be range restricted, which means that every **Xi** occurs in Q. This assumption is made in order to guarantee that only fully instantiated answers are returned.

The reordering of subgoals intends to improve efficiency. In order to be able to measure efficiency, one needs to have some kind of measuring unit. The most accepted one for software is CPU-time; but since this is too low level and therefore too untransparent for implementation ends, it is useful to adopt a more conceptual unit: the unification operation. The matching of two terms costs as many unification operations as the number of variables or constants that are matched. For example, matching `S(a,b)` and `S(X1,c)` has cost 4 (since fact tuples cannot contain functors, the problem of matching variables or constants with terms does not arise). A condition is interpreted as having a cost, equal to the number of variables it contains, e.g. `S > 30` has cost 1 (note that this cost measurement is rather arbitrary chosen).

In what follows we will first deal with the case that every subgoal is relational. Later on we will handle the other possibilities.

Handling Relational Subgoals
Suppose we have two relations, S and T, and a query:

S	
a	b

T	
c	b

1 There are Prolog implementations, like Arity Prolog, that offer more storage control.

2 Appendix 2 elabotes the numbers and formulas as they appear in this subsection.

```
c     d          a     b
c     e          c     e
f     g          a     g
-------------------------
:- s(X1,X2), t(c,X2)
```

Executing this query will require $4 \times (1 + 4) = 20$ match-operations (every matching of two atoms is called a match-operation). Evaluating the subgoals in reverse order on the other hand, will require only $2 \times (1 + 4) + 2 = 12$ operations. The reason for gaining this much is clear: if we evaluate `t(c,X2), s(X1,X2)`, then for every tuple in t that has no **c** as its first argument, the second subgoal will not be evaluated. Stated differently, we should aim at making Prolog backtrack as soon as possible.

The consequence of executing the most instantiated subgoal first however, is not the only factor that influences the processing of a query. We should therefore develope a formula that takes into account all factors, giving an exact answer in terms of unification operations.
The total cost of executing a query consisting of **n** relational subgoals can be stated as

$$\textbf{Total Cost} = \sum_{k=1}^{a_1} \left(\left\{O_{T_1}^{k} - O_{T_1}^{k-1}\right\} + \sum_{k=1}^{a_2} \left(\left\{O_{T_2}^{k} - O_{T_2}^{k-1}\right\} + \ldots + \sum_{k=1}^{a_n} \left(\left\{O_{T_n}^{k} - O_{T_n}^{k-1}\right\}\right) \ldots\right)\right)$$

where T_k is a relational subgoal, having index k in the query (this formula is derived in appendix 2),
a_k is the total number of answers to T_k, making allowance for possible instantiations of some of its arguments that result from evaluating T_1 to T_{k-1} and
O is the cost of matching two atoms.
From this formula, we can determine the factors that effect the execution cost:
1) The number of arguments of each T, because O partly depends on it.
2) The size of a_1 to a_n. Every a_k in turn, is determined by
a) the cardinality of T_k,
b) the ratio uninstantiated/instantiated variables and
c) the number of tuples to which the instantiated variables refer.
It should be noted that 1 and 2b are purely syntactical standards, while 2a and 2c are dependent on the current extension of the underlying relations.
Because of the nesting of summation signs, it seems wise to see to it that the T_k's are arranged in ascending order according to their a_k's. The drawback of using the formula however, is that every a_k is a function of a_1 to a_{k-1}(on the ground of 2b and 2c). This means that the only way to compute the optimal ordering, is to consider every possible permutation. For, say, 8 subgoals, the number of permutations jumps to over 40,000 (8!), so this method can be prohibitively expensive.

Therefore KBMS5 relies on a heuristic formula that uses all of the four above mentioned factors, without regarding the mutual influences: the Current Cost (CC). Described in words, this formula says: the cost of evaluating a subgoal is equal to the number of expected instantiations for it, times the number of its arguments.
a) $CC_k = M_k$ if the primary key arguments of subgoal T_k are all instantiated, which means that at most one answer will be found,

b) $CC_k = C_k \times (1 - \frac{I_k}{M_k}) \times M_k$ otherwise.

where M_k is the number of arguments of T_k, C_k is the cardinality of relation T_k and I_k denotes the number of instantiated arguments at the time subgoal T_k is evaluated; so $C_k \times (1 - \frac{I_k}{M_k})$ expresses the estimated number of answers to the subgoal T_k and
C_k accounts for the influence of factor 2a,
$\frac{I_k}{M_k}$ accounts for 2b,
M_k in b) accounts for 1 and

M_k in a) for 2c (partly).
Of course one could wonder why the factors in formula b) are multiplied with each other. The reason is, that the current cost is proportional to every factor. If the cardinality or the number of arguments increases, or the ratio between the number of instantiated and uninstantiated arguments decreases, the expected absolute cost of evaluating the subgoal will increase. However, the factors are not multiplied with coefficients, since we are not interested in an absolute cost, but rather in a relative measuring unit.

Part a) of the formula could be refined by not only looking at primary key arguments, but storing statistical information about every relation as well; for example, the number of expected answers for every combination of instantiated arguments. In [Li84] (pp. 118-125), Li uses a domain statistic, that is, the size of the domain over which an argument can range. Besides, he places cuts between subgoals in order to improve efficiency, which is an interesting but dangerous improvement, since it may quite drastically affect the way in which the query is evaluated.
The strategy adopted in KBMS5 then, is based on the use of the current cost and is a very simple one:

1) Compute CC for every subgoal.
2) Process the cheapest.
3) Go to 1) for the remaining subgoals.

It should be stressed that this strategy is not necessarily optimal, but heuristic.

Handling Implicit Relational Subgoals
The former leaves us with the cases where a subgoal is either conditional or implicit relational. In the latter case, we can act in one of the two following ways in order to compute CC.

1) Pre-compile the query into a list of only relational and conditional subgoals. This means that every implicit subgoal is replaced by the body of its corresponding clause(s). Since this body is defined as being a query itself, the replacement process can be recursive.
Unfortunately, a problem arises, stemming from the fact that we're working at the meta-level:

- In the case of recursive deductive laws there is no way for the compiler to choose between the stopping condition and the recursive rule(s) without actually calling the subgoal. For example, consider the database

```
A            B

2            1    2
a(X) :- b(X,Y), a(Y).
```

The query `a(Y)` could be replaced by itself, but also by `b(X,Y),a(Y)` or `b(X,Y), b(X,Y), a(Y)` etcetera.

There seems to be no way to circumvent this problem other than by lifting all of the deduction process of Prolog to the meta-level. The efficiency loss then would certainly not warrant the improved control.

2) Store the cardinality of the implicit relations as well as their primary key arguments. When optimising the query, treat the implicit subgoals the same way as the explicit ones.
Again, note some drawbacks:

- The value of C_k is no longer a reliable measure for determining CC_k. Consider the following example:

```
B            C
```

```
I          I
2          2
3
-------------
a(X) :- c(X).
```

The (derived) cardinality of A is 2, while C_b is 3. The call `a(X)` however, is twice as expensive as the call `b(X)`.

- To compute the cardinalities of the implicit relations, we must call every deductive law of which the body contains T_k, whenever a tuple T_k is inserted, deleted or updated.
Now suppose we have the following database:

```
B

3     I
-------------
a(X) :- b(X,1).
```

and a program for inserting a tuple into it:

```
insert(Tuple) :-                % When we ask 'insert(b(4,1))'
      assert(Tuple),            % we get:
      clause(Head,Body),        % Head = a(X); Body = b(X,1),
      member(Tuple,Body),       % true for X = 4 !,
      count(call(Head),C).[1]   % count(call(a(4)),C)
```

C will be 1, which is incorrect. Again, the instantiation mechanism is to blame.

- Determining the primary key arguments is not easy; since we record these arguments during database design however, this problem is not a real one in KBMS5.

In spite of its weaknesses, KBMS5 adopts a (simplified) instance of this latter strategy, mainly in order not to overload the optimization process. This means that the optimiser now does not need to have any knowledge about implicit relations. The cardinality and primary key arguments are simply read off from the 'relation'-clauses.

Handling Conditional Subgoals

As for the conditional subgoal, this can be seen as a special kind of instantiation. While a conventional instantiation restricts the number of possible values of an argument to one, a condition restricts it to a set of possible domain values. As an example of the congruence of these two concepts, consider the query

```
:- employee(art,Group) .
```

that can be rewritten as

```
employee(Name,Group), Name = art.
```

Ideally therefore, the computation of CC_k should involve some measure for the number and kind of conditions that restrict the arguments of T_k. On the one hand however, this means that for every relation, we would need an enormous amount of statistical information; on the other hand, using

1 *Count(call(Head),C)* is not meant to be a Prolog goal.

such a measure would reintroduce the connected consideration of the subgoals. For these reasons, the reordering of a query in KBMS5 is based solely on relational subgoals. A condition is placed in the reordered query as soon as all its variables have occurred in this query (this condition must be imposed, in order to prevent evaluation of an uninstantiated condition). The rationale underlying this method is that placing conditions this way will make the query at least as good, and probably much better.

Handling General Queries

Summarizing, the reordering strategy is as follows:

(1) Compute the current cost for every non-conditional subgoal.
(2) Put the cheapest one of the initial query at the end of the ordered query.
(3) Append to this list all the conditions whose variables occur in the ordered query.
(4) Go to (1) for the remaining query.

Remarks and Conclusions

Anyone working with KBMS5 will notice that this program is very far from being useful in practice. On one hand, this is caused by practical considerations. It was not intended to build a commercial system, so little or no attention was paid to topics like user-interfacing and security. Furthermore, since the research on deductive databases has yielded a vast number of subjects, time was too limited to be able to cover all interesting and useful features.
On the other hand, several drawbacks arise from the concept of deductive databases as such, and from the use of Prolog for building a DDBMS.

In what follows, some practical and theoretical problems are discussed, as well as suggestions for solving them.

1. Improvements to KBMS5

In this section, four major limitations of KBMS5 are described. These limitations are

A) **Performance is poor.**
For every database transaction on a reasonably large database, transaction time is far too long:

- data retrieval is slow, since data cannot be organized well. Furthermore, in spite of query optimization, retrieval suffers from the fact that backtracking is difficult to control and therefore traverses many redundant paths;

- updating is slow, since all deductive laws and integrity constraints have to be checked, in order to see which ones are applicable; again this can be viewed as a problem of physical data organization.
We should however, be careful in using the word 'slow'. It is simply not fair to compare deductive database updating performance with the updating performance of a conventional system. Besides, it is possible to trade off retrieval time against query time (although this would probably not shorten average response time).
The problems in data retrieval are more serious, since they are mainly caused by the Prolog language itself; these problems are at present prohibitive for building large databases. Improvements to Prolog might be

* intelligent backtracking control mechanisms other than the cut. A good example of this is found in [Nais84] where a resolution strategy called *Heterogeneous SLD-Resolution* is discussed. Its computation rule does not make it backtrack to the most recent backtracking point (= a node in the search tree that has several children), but rather to the point where the wrong path was chosen.

* mechanisms that enable one to organize clauses, for example in an indexed file.

B) **Only limited query facilities are provided**. This problem applies to user communication, for example query languages, as well as to the way queries are evaluated. Various improvements are imaginable:

* design a richer query language, containing functions like 'count', 'average', 'group' etcetera. These improvements are easy to implement, as noted in [Lloy83a]. Furthermore, the query language should be more user-friendly; for example, in KBMS5 the user needs to remember the number and order attributes of a relation.

* supply the user with more intelligent answers. That is, the answer to a query might not necessarily consist of facts, but also of rules. For example, if we ask 'who earns at least 40,000' the answer might be 'all managers do'.

Another possibility is the adding of useful extra information to the answer. For example, if we ask for the departure time of a flight from London to Paris before 7 A.M. and there is no such flight, the system might provide alternative times or routes.

On these topics, see [Gall88], p.62 for an overview and [Imie88] for a method that allows for rule-based answers.

* incorporate more semantic knowledge into the query optimization process. KBMS5 only uses primary key and cardinality information to estimate the number of answers to a subgoal. Semantic optimization on the other hand, relies on the knowledge that integrity constraints restrict the number of possible extensions of relations. This implies that we can determine impossible answers. Thus, the query evaluation process can be forced not to search for unfindable solutions. For example, if we have a constraint

```
:- father(art,X).          % (art has no children)
```

and a query

```
?- father(john,Child), father(Child,Grandchild).
```

we know that if `Child` is equal to `art` the second subgoal will yield no answers. So we rewrite the query to

```
?- father(john,C), not C = art, father(C,Gc).
```

In [Chak88] a method for semantic optimization, based on a complex technique, called 'subsumption', is discussed.

C) **No support is offered to the user for creating databases**. KBMS5, and in fact any current database system, assumes that the database is a correct reflection of a universe of discourse. That is, no attention is paid to the process of conceptual modelling and designing data structures. While the modelling process is still left to the user, suggestions for mapping a conceptual schema directly onto a database structure, are provided by [Dart88] and [Grat87]. Both articles rely on NIAM ([Verh82]) for specifying the model. Part of the NIAM primitives are represented directly in first order logic. However, it remains questionable whether these methods lead to reasonably efficient implementations. Furthermore, no use seems to be made of specific deductive database capabilities like deduction of information.

Other, more concrete improvements to KBMS5, are quite easy to implement. Examples are the automatic generation of common integrity constraints like primary keys (we might simply state 'Name of an employee is primary' instead of 'Group = Group1 :- employee(Name,Group), employee(Name,Group1)'), and the automatic recording of a type of an integrity constraint.

D) **It is unfeasible to build large databases**. This is mainly due to the fact that the entire database is kept in main memory. To overcome this problem, the most attractive alternative seems to be to interface the Prolog system with a conventional retrieval facility. This would imply that all the deduction steps following a query are performed by Prolog. When nothing but facts have to be retrieved, the conventional system takes over (perhaps after the query is translated into a relational calculus expression). This method is called the 'compiled approach' in [Gall84]. [1] ([Lloyd83a])

1

Lloyd ([Lloy83a]) points out that an application has been built, based on this idea, that takes up about 260 Megabytes!

References

[Brat86] I. Bratko, **Prolog Programming for Artificial Intelligence**, Addison-Wesley, 1986

[Brod84] M.L. Brodie, J. Mylopoulos and J.W. Schmidt (eds.), **On Conceptual Modelling**, Springer, 1984

[Brod86] M.L. Brodie and J. Mylopoulos, **Knowledge Bases and Databases: Semantic vs. Computational Theories of Information**, in: G. Ariav and J. Clifford (eds.), New Directions For Database Systems, Ablex, 1986

[Chak88] U.S. Chakravarthy, J. Grant and J. Minker, **Foundations of Semantic Query Optimization for Deductive Databases**, in : [Mink88]

[Clar78] K.L. Clark, **Negation as Failure**, in: [Gall78]

[Dart88] P.W. Dart and J. Zobel, **Conceptual Schemas Applied To Deductive Databases**, Information Systems Vol. 13, No. 3, pp. 273-287, 1988

[Date86] C.J. Date, **Relational Database: Selected Writings**, Addison-Wesley, 1986

[Li84] Deyi Li, **A PROLOG Database System**, Research Studies Press, 1984

[Fros86] R.A. Frost, **Introduction to Knowledge Base Systems**, Collins, 1986

[Gall78] H. Gallaire and J. Minker (eds.), **Logic and Databases**, Plenum Press, 1978

[Gall84] H. Gallaire, J. Minker and J-M Nicolas, **Logic and Databases: A deductive approach**, Computing Surveys, Vol. 16, No. 2, pp. 170-180, June 1984

[Gall88] H. Gallaire and J-M Nicolas, **Deductive Databases**, Tutorium der ECAI - 88, 2./3.8.1988, Munich

[Gene87] M. Genesereth and N. Nilsson, **Logical Foundations of Artificial Intelligence**, Morgan Kaufmann Publishers, 1987

[Grat87] G.M. McGrath, **The Transition to Fifth Generation Technology: Conceptual Scheme Implementation**, The Australian Computer Journal, Vol. 19, No. 1, pp. 16-23,February 1987

[Imie88] T. Imielinski, **Intelligent Query Answering in Rule Based Systems**, in: [Mink88]

[Kent78] W. Kent, **Data and Reality**, North-Holland, 1978

[Lloy83a] J.W. Lloyd, **An Introduction to Deductive Database Systems**, The Australian Computer Journal, Vol. 15, No. 2, pp. 54-57, May 1983

[Lloy83b] J.W. Lloyd and R.W. Topor, **Making Prolog More Expressive**, The Journal of Logic Programming 1, pp. 225-240, May 1983

[Maie83] D. Maier, **The Theory of Relational Databases**, Computer Science Press, 1983

[Malp87] J. Malpas, **Prolog: A Relational Language and its Applications**, Prentice-Hall, 1987

[Mink88] J. Minker (ed.), **Foundations of Deductive Databases and Logic Programming**, Morgan Kaufmann, 1988

[Nico78a] J-M Nicolas and H. Gallaire, **Database: Theory vs. Interpretation**, in: [Gall78]

[Nico78b] J-M Nicolas and K. Yazdanian, **Integrity Checking in Deductive Databases**, in: [Gall78]

[Nico79] J-M Nicolas, **A property of logical formulas corresponding to integrity constraints on database relations**, in: Proceedings of the Workshop on Formal Bases for Data Bases, Toulouse, 1979

[Reit78] R. Reiter, **Deductive Q-A on Relational Databases**, in: [Gall78]

[Reit86] R. Reiter, **Towards a Logical Reconstruction of Relational Database Theory**, in: [Brod86]

[Sadr88] F. Sadri and R. Kowalski, **A Theorem-proving Approach to Database Integrity**, in: [Mink88] Ch. 9

[Ster86] L. Sterling and E. Shapiro, **The Art of Prolog**, The MIT Press, 1986

[Topo85] R.W. Topor, T. Keddis and D.W. Wright, **Deductive Database Tools**, The Australian Computer Journal, Vol.17, No. 4, pp. 163-172, November 1985

[Ullm85] J.D. Ullman, **Implementation of Logical Query Languages for Databases**, ACM Transactions on Database Systems, Vol. 10, No. 3, September 1985

[Verh82] G.M.A. Verheijen and J. van Bekkum, **NIAM: An Information Analysis Method**, in: Olle, Sol and Verrijn-Stuart (eds.), Information Systems Design Methodologies, North Holland, 1982

appendix 1 An Example of Query Optimization

In section 4.2. an optimization strategy for queries was developed. In this appendix an example of how this strategy works is described.
Consider the example database that comes with KBMS5, called 'sportdb5'. It contains information on a sports club, contained in 4 base relations called `player` (14 tuples), `team` (2 tuples), `match` (10 tuples) en `penalty` (8 tuples).
With respect to this database we might pose the following query:

```
q([S,N,B,% give me number, name and penalty
player(S,N,_,_),% of all players
team(_,S,_),% who are captain of some team and
penalty(_,S,B),% who have had a penalty
B  > 49]).% of at least 49,-
```

Now recall that the Current Cost was defined to be
$CC_k = M_k$ if all primary arguments of T_k are instantiated;
$CC_k = M_k \times (1 - \frac{I_k}{M_k}) \times M_k$ otherwise.
According to the strategy as described in the last paragraph of 4.2. the reordering passes off as follows:

1) The reordered query is empty, written as [ ];
2) CC(`player(S,N,_,_)`) = $14 \times (1 - \frac{0}{1}) \times 4 = 56$;
CC(`team(_,S,_)`) = $2 \times (1 - \frac{0}{1}) \times 3 = 6$;
CC(`penalty(_,S,B)`) = $8 \times (1 - \frac{0}{1}) \times 3 = 24$;
3) The reordered query becomes `[team(_,S,_)]`;
4) Now `S` is instantiated with, say, `s`;
5) CC(`player(s,N,_,_)`) = 4 since `s` is a primary key value;
CC(`penalty(_,s,B)`) = 3 for the same reason;
6)The reordered query becomes `[team(_,S,_), penalty(_,S,B), player(S,N,_,_)]`;
7)The condition `B > 49` is placed behind `penalty(_,S,B)` since at this point `B` becomes instantiated;
8)The query to be evaluated now is
`[team(_,S,_), penalty(_,S,B), B > 49, player(S,N,_,_)]`.

In an experiment the unordered and ordered query required an evaluation time of 12 and 5 respectively. This implies an improvement of over 40%! However, for queries on a database of this size, the improvement is entirely used up by the reordering process.

appendix 2 Elaboration of Formulas in II.4.2.

The section about query optimization shows some numbers and formula's concerning query evaluation. Since it is not made clear how these statements are arrived at, this appendix will elaborate them.

1) Suppose we have two relations, S and T, with the following extensions:

S	
a	b
c	d
c	e
f	g

T	
c	b
a	b
c	e
c	g

The execution trace of the query `S(X1,X2), T(c,X2)` is as follows (every '&' denotes a match-operation):

```
S(X1,X2) & S(a,b) :     T(c,b) & T(c,b) : answer
                        T(c,b) & T(a,b) fail
                        T(c,b) & T(c,e) fail
                        T(c,b) & T(a,g) fail
S(X1,X2) & S(c,d) :     T(c,d) & T(c,b) fail
                        T(c,d) & T(a,b) fail
                        T(c,d) & T(c,e) fail
                        T(c,d) & T(a,g) fail
S(X1,X2) & S(c,e) :     T(c,e) & T(c,b) fail
                        T(c,e) & T(a,b) fail
                        T(c,e) & T(c,e) : answer
                        T(c,e) & T(a,g) fail
S(X1,X2) & S(f,g) :     T(f,g) & T(c,b) fail
                        T(f,g) & T(a,b) fail
                        T(f,g) & T(c,e) fail
                        T(f,g) & T(a,g) fail
```

So this query will cost $4 \times (1 + 4) = 20$ match-operations. Evaluating the reverse query, `T(c,X2), S(X1,X2)` leads to:

```
T(c,X2) & T(c,b)  :     S(X1,b) & S(a,b) : answer
```

```
                          S(X1,b) & S(c,d) fail
                          S(X1,b) & S(c,e) fail
                          S(X1,b) & S(f,g) fail
T(c,X2) & T(a,b) fail
T(c,X2) & T(c,e)   :      S(X1,e) & S(a,b) fail
                          S(X1,e) & S(c,d) fail
                          S(X1,e) & S(c,e) : answer
                          S(X1,e) & S(f,g) fail
T(c,X2) & T(a,g) fail
```

This evaluation sequence costs only 2 × (1 + 4) + 2 = 12 operations. This is of course due to the times the first subgoal fails.

2) The formula for calculating the total cost of executing a query consisting of n relational subgoals can be derived as follows.
A query is a list of subgoals $T_1 \ldots T_n$, all refering to relations in the database; the cost of retrieving an answer is equal to the total number of match-operations needed.
Say i is an index that denotes the physical rank of a fact in the extension of some relation; for example, i = 2 for `S(c,d)`. The first answer to the first subgoal is found at location $i^1_{T_\cdot}$: the value of this index is equal to the total number of match-operations. The additional cost for retrieving the second answer to this subgoal is $(i^2_{T_\cdot} - i^1_{T_\cdot})$. This goes on until the last answer to the subgoal is found with cost $(i^a_{T_\cdot} - i^{a-1}_{T_\cdot})$, where a denotes the total number of answers.
Of course an answer to subgoal T_k might be dependent on the instantiations for every one of subgoals $T_1 \ldots T_{k-1}$. Thus we can write, for example

$$(\{i^1_{T_\cdot} + \ldots + (i^a_{T_\cdot} - i^{a-1}_{T_\cdot})\})^{T_\cdot}$$

This formula expresses the total cost of finding all answers to subgoal T_n, related to the first answer found in T_2. The total cost, being equal to the total number of match-operations needed, then becomes

$$i^1_{T_\cdot} + [i^1_{T_\cdot} + (\ldots(i^1_{T_\cdot} + \ldots + \{i^a_{T_\cdot} - i^{a-1}_{T_\cdot}\})^{T_\cdot} \ldots) + \ldots + \{i^a_{T_\cdot} - i^{a-1}_{T_\cdot}\} +$$
$$+ (\ldots(i^1_{T_\cdot} + \ldots + \{i^a_{T_\cdot} - i^{a-1}_{T_\cdot})^{T_\cdot} \ldots)] + \ldots \{i^a_{T_\cdot} - i^{a-1}_{T_\cdot}\} + [\ldots.]$$

The transition to the cost formula based on unification operations is quite simple. We define O_k to be i_k times the number of arguments of subgoal T_k. Next we group all expressions concerning a single subgoal together, getting

$$\sum_{k_\cdot=1}^{a_\cdot}(\{O^k_{T_\cdot} - O^{k-1}_{T_\cdot}\} + \sum_{k_\cdot=1}^{a_\cdot}(\{O^k_{T_\cdot} - O^{k-1}_{T_\cdot}\} + \ldots + \sum_{k_\cdot=1}^{a_\cdot}(\{O^k_{T_\cdot} - O^{k-1}_{T_\cdot}\})\ldots))$$

where all $O^0_T = 0$ and $a_n = 0$ if the corresponding call T_{k-1} fails.

3) The current cost is defined to be the number of expected answers to a subgoal times its number of arguments:

$$CC_k = \tilde{A}_k \times M_k$$

Obviously, CC_k is positively correlated with the cardinalityC_k of the subgoal. Furthermore, a linear relation is assumed with the number of instantiated arguments. For example, `T(X1,X2)` returns

C_k answers, whereas `T(X1,c)` returns $\frac{1}{2} C_k$ answers. A different situation arises if the primary key arguments are all instantiated, for then at most one answer can be returned, so $\tilde{A}_k$ is 1. Thus,

$\tilde{A}_k = 1$ if all primary key arguments C_k are instantiated;

$\tilde{A}_k = C_k \times (1 - \frac{I_k}{M_k})$ otherwise

where $I_k \equiv$ number of instantiated arguments and
$M_k \equiv$ total number of arguments.

The current cost now is easy to formulate as

$CC_k = M_k$ if $\tilde{A}_k$ is 1;

$CC_k = C_k \times (1 - \frac{I_k}{M_k}) \times M_k = C_k \times (M_k - I_k)$ otherwise.

appendix 3 Listing of the Program

In this appendix the source code of KBMS5 is presented. Most of the user communication commands are omitted as well as much of the code that is not needed for understanding the program.

```
start :-                                    % initializing windows etc.;

stop :-                                     % closing windows;

s(Database) :-                              % saving database contents
      tell(Database),
      forall(retract(relation(A,B,C,D)),
      (writeq(relation(A,B,C,D)),write('.~J~M'))),
      forall(retract(dlaw(E)),(write(dlaw(E)),write('.~J~M'))),
      forall(retract(ic(F,G)),(write(ic(F,G)),write('.~J~M'))),
      forall(retract(tuple(H,I,J)),
      (write(tuple(H,I,J)),write('.~J~M'))),
      forall(clause(query(Name),A),
      (write((query(Name) :- A)),write('.~J~M'))),
      kill(query),
      write(bottom), write('.~J~M'),
      told,
      kill(database), assert(database(empty)).

l(Database) :-                              % loading database contents
      see(Database),
      retrieve,
      seen.
retrieve :-
      read(C),
      assert(C),
      C = bottom,
      retract(C), kill(database), assert(database(full)).
retrieve :-
      retrieve.
```

```
database(empty).                          % this clause is updated
                                          % when a database is loaded

% ============================
% DATABASE MANIPULATION MODULE
% ============================

i :-                                      % shows information about
                                          % database contents

c :-                                      % commit command; checks for
                                          % constraints for which no
                                          % revalidation is possible
      ic((C :- A),type2a),
      tuple(insert+N,0,Tuple),
      member(Tuple,A),
      evaluate(A,i),
      not C,
      show(['Constraint violation:',(C :- A)]),
      r.
c :-                                      % if the database is
                                          % consistent, all changes
                                          % are committed
      consistent,
      forall(retract(tuple(delete-N,N,Tuple)),inc(Tuple,-1)),
      forall(retract(tuple(delete-N1,C,Tuple)),
      (N1 is N-1, assert(tuple(accept,N1,Tuple)),inc(Tuple,-1))),
      forall(retract(tuple(insert+N2,C1,Tuple)),
      (N3 is N2+C1,
      assert(tuple(accept,N3,Tuple)),inc(Tuple,1))).

inc(Tuple,I) :-                           % the occurrence number of
                                          % a tuple is adapted

r :-                                      % all changes are rolled
                                          % back
      retractall(tuple(insert+N,0,Tuple)),
      forall(retract(tuple(insert+N1,C,Tuple)),
      assert(tuple(accept,C,Tuple))),
      forall(retract(tuple(delete-0,0,Tuple)),
```

```
        assert(tuple(accept,1,Tuple))),
        forall(retract(tuple(delete-N2,C1,Tuple)),
        assert(tuple(accept,C1,Tuple))),

consistent :-                                   % the database is consistent
                                                % if both bagof commands have
                                                % no solutions
        bagof1(_,(tuple(insert+N,0,Tuple),
                  invalidate(insert,Tuple,Ic,Type,Ic1),
                  revalidate(insert,Tuple,Ic,Type,Solution),
                  show(['Violation of',Ic1,'~J~M (',Ic,')~J~M',
                  'Suggested solution: ']),
                  show(Solution), nl, nl),
                  L1),
        bagof1(_,(tuple(delete-N1,_,Tuple),
                  invalidate(delete,Tuple,Ic,Type,Ic1),
                  revalidate(delete,Tuple,Ic,Type,Solution),
                  show(['Violation of',Ic1,'~J~M (',Ic,')~J~M',
                  'Suggested solution: ']),
                  show(Solution), nl),
                  L2),
        L1 = [], L2 = [].

invalidate(insert,Tuple,(C :- A),Type,Ic) :-
                                                % determines which constraint
                                                % is violated by inserting a
                                                % tuple
        ic((C :- A),Type),
        not violated((C :- A)),
        assert(old_to_new((C :- A))), retract(old_to_new(I)),
        member(Tuple,A),
        evaluate(A,i),
        not evaluate(C,i),
        toground(I,Ic), !.
invalidate(delete,Tuple,(C :- A),Type,Ic) :-
                                                % determines which constraint
                                                % is violated by deleting a
                                                % consequent tuple
        ic((C :- A),Type),
        not violated((C :- A)),
```

```
      assert(old_to_new((C :- A))), retract(old_to_new(I)),
      Tuple = C,
      evaluate(A,i),
      toground(I,Ic), !.
invalidate(delete,Tuple,(C :- A),Type,Ic) :-
                                          % determines which constraint
                                          % is violated by deleting a
                                          % antecedent tuple
      ic((C :- A),Type),
      not violated((C :- A)),
      assert(old_to_new((C :- A))), retract(old_to_new(I)),
      member(not Tuple,A),
      evaluate(A,i),
      not evaluate(C,i),
      toground(I,Ic), !.

                                          % determines solutions to
                                          % various violations
revalidate(insert,Tuple,(C :- A),type1a,[insert,C]).
revalidate(delete,Tuple,(C :- A),type1a,[delete,A]).
revalidate(delete,Tuple,(C :- A),type2a,['delete a tuple
from',S]) :-
      bagof(T,(member(T,A),T =.. [R|_],relation(R,_,_,_),not on(not
Tuple,[T])),S).
revalidate(insert,Tuple,(C :- A),type2b,['delete a tuple
from',S]) :-
      bagof(T,(member(T,A),T =.. [R|_],relation(R,_,_,_),not T =
      Tuple),S).
revalidate(insert,Tuple,(C :- A),type1b,[insert,C,or|Y]) :-
      revalidate(insert,Tuple,(C :- A),type2b,Y).
revalidate(delete,Tuple,(C :- A),type1b,S) :-
      revalidate(insert,Tuple,(C :- A),type2b,S).

u(Old,New) :-                             % updating is deleting +
                                          % inserting
      d(Old),
      i(New).

i(Tuple) :-                               % inserting an incorrect
                                          % tuple
```

```
      incorrect(Tuple,Errorcode),
      message(Errorcode).
i(Tuple) :-                                  % correct insertion
      i1(Tuple).

i1(Tuple) :-
      i2(Tuple,0).
i1(Tuple) :-
      write('Transactions completed.~J~M').

i2(Tuple,Tab) :-                             % inserting a tuple proposed
                                             % for deletion
      retract(tuple(delete-0,C,Tuple)),
      assert(tuple(insert+1,C,Tuple)), nl, !,
      insert_consequents(Tuple,Tab).
i2(Tuple,Tab) :-                             % inserting a tuple proposed
                                             % for deletion
      retract(tuple(delete-N,C,Tuple)),
      N1 is N - 1, nl,
      assert(tuple(delete-N1,C,Tuple)), !, fail.
i2(Tuple,Tab) :-                             % inserting a tuple proposed
                                             % for insertion
      retract(tuple(insert+N,C,Tuple)),
      N1 is N + 1,
      assert(tuple(insert+N1,C,Tuple)),
      !, fail.
i2(Tuple,Tab) :-                             % inserting a tuple that did
                                             % not exist before
      assert(tuple(insert+1,0,Tuple)), nl,
      insert_consequents(Tuple,Tab).

insert_consequents(Tuple,Tab) :-             % determining the derived
                                             % informations of a tuple
      dlaw((Consequent :- Antecedents)),
      member(Tuple,Antecedents),
      evaluate(Antecedents,i),
      i2(Consequent,Tab1).

d(Tuple) :-                                  % deleting module
      tuple(_,_,Tuple),
```

```
      to_terminal([Tuple],[],Dlist),
      d1(Dlist),
d(Tuple) :-
      fail.

d1([]).
d1([X|Y]) :-                                % forward chaining from
                                            % a set of terminal tuples
      d2(X,0);
      d1(Y).

d2(Tuple,Tab) :-                            % forward chaining for a
                                            % single tuple
      retract(tuple(_,_,Tuple)),
      assert(tuple(delete-0,0,Tuple)), !,
      delete_consequents(Tuple,Tab);
      true.

delete_consequents(Tuple,Tab) :-            % determining the derived
                                            % information of a tuple
      tuple(delete-C,C,Tuple),
      dlaw((Consequent :- Antecedents)),
      member(Tuple,Antecedents),
      evaluate(Antecedents,d),
      not tuple(delete-C1,C1,Consequent),
      decrement(Consequent),
      delete_consequents(Consequent,Tab1).

                                            % backward chaining from
                                            % the initial tuple
to_terminal([Tuple],[],[Tuple]) :-
      not bagof(_,(dlaw((Tuple :- A)),evaluate(A,i)),L),
to_terminal(Inlist,Previous_outlist,Outlist) :-
      expand(Inlist,Helplist1),
      harmonica_sort(Helplist1,Helplist2),
      not Helplist2 = Previous_outlist,
      and_or(Helplist2),
      choose(Helplist2,Helplist3),
      to_terminal(Helplist3,Helplist3,Outlist).
to_terminal(L,P,L) :-
```

```
      show(['~J~M Set of terminal tuples to
      be deleted:',L,'~J~M']).

                                          % one step in the backward
                                          % chaining process
expand([],[]).
expand([X|Rest],[Expanded|Rest1]) :-
      bagof(O,(dlaw((X :- A)),evaluate(A,i),
      expand(Rest,Rest1).
expand([X|Rest],[X|Rest1]) :-
      expand(Rest,Rest1).

choose(..) :-                             % choosing between chaining
                                          % paths


and_or(..) :-                             % writing down possible
                                          % deletion candidates

or(..) :-                                 % writing down disjunctive
                                          % candidates

decrement(..) :-                          % decrementing the occurence
                                          % number of a tuple

brackets((X,Y)).

evaluate((X,Y),S) :-                      % evaluating a list of
                                          % subqueries
      evaluate(X,S),
      evaluate(Y,S).
evaluate(not X,S) :-                      % evaluating a negated
                                          % relational subquery
      not evaluate(X,S), !.
evaluate(X,i) :-                          % evaluating a relational
                                          % subquery
      is_a(X,relation),
      tuple(Status,Counter,X), not Status = delete - Counter.
evaluate(X,d) :-                          % evaluating a relational
                                          % subquery
      is_a(X,relation),
      tuple(_,_,X).
```

```
evaluate(X,_) :-                          % evaluating a condition
      not brackets(X),
      not is_a(X,relation),
      call(X).

member(..) :-

harmonica_sort(X,Y) :-
      harmonica(X,Z),
      sort(Z,Y).
harmonica(..) :-                          % getting rid of brackets

bagof1(X,Y,Z) :-                          % bagof that accepts empty
                                          % solution lists
      bagof(X,Y,Z), !.
bagof1(X,Y,[]).

violated.


% ============
% QUERY MODULE
% ============

sq(Query) :-                              % asserting a standard query
      read(Name),
      assert((query(Name) :- q(Query))), !,
      q(Query).

q(Query) :-                               % query module
      test(process(Query)),
      optimal(Query),

optimal(Query) :-                         % optimizing a query
      toground(Query,G,Gtotal),
      split(G,Gquery,Gconditions,Garguments),
      optimize(Gquery,Gconditions,O),
      concat(O,Garguments,Optimized_gquery),
      tohollow(Optimized_gquery,Optimized_query,Gtotal),
      test(process(Optimized_query)).

split(..) :-                              % splitting a query in
                                          % a grouded query, a set of
```

```
                                        % grounded conditions and
                                        % a set of variables that
                                        % appear in it

process(Q) :-                           % evaluation module
      split1(Q,Arguments,Query),
      forall(process1(Arguments,Query),(show(Arguments),nl)).

split1(..) :-                           % splitting a query in a
                                        % proper query and a set of
                                        % selected attributes

process1(Arguments,[]).
                                        % evaluating a relational
                                        % query
process1(Arguments,[Subquery|Rest]) :-
      is_a(Subquery,relation),
      tuple(Status,Counter,Subquery),
      process1(Arguments,Rest).
                                        % evaluating a condition
process1(Arguments,[Subquery|Rest]) :-
      not is_a(Subquery,relation),
      call(Subquery),
      process1(Arguments,Rest).

                                        % optimization module
optimize(Query,Conditions,Optimized_query) :-
      reorder(Query,Sorted_query,[]),
      insert_conditions(Conditions,Sorted_query,Optimized_query).

reorder([],[],I).                       % reordering a query
reorder(Query,[Cheapest|Rest],I) :-
      minlist(Query,Cheapest,I),
      Cheapest =.. [F|A],
      concat(I,A,I1),
      remove(Cheapest,Query,Query1),
      reorder(Query1,Rest,I1).

minlist([X],X,I).
minlist([X,Y|Rest],Min,I) :-            % determining the cheapest
                                        % query at this point
```

```
        minlist([Y|Rest],Minrest,I),
        cost(X,I,Cost1),
        cost(Minrest,I,Cost2),
        min(X,Minrest,Cost1,Cost2,Min).

min(..) :-                                  % determining the minimum
                                            % of two costs

                                            % determining the cost of
                                            % a subquery
cost(Subquery,Instantiated_variables,Cost) :-
        Subquery =.. [Predicate|Arguments],
        relation(Predicate,_,Indices,_),
        primary_key(Indices,Arguments,P_key),
        subset(P_key,Instantiated_variables),
        length(Subquery,Cost).
cost(Subquery,Instantiated_variables,Cost) :-
        Subquery =.. [Predicate|Arguments],
        uninstantiated(Arguments,Instantiated_variables,M_minus_I),
        relation(Predicate,_,_,C),
        Cost is C * M_minus_I.

subset(..) :-

primary_key(..) :-                          % determining the primary key
                                            % arguments of a subquery

uninstantiated(..) :-                       % counting the number of
                                            % uninstantiated arguments

remove(..) :-

                                            % inserting conditions into
                                            % the reordered query
insert_conditions([],Z,Z).
insert_conditions([X|Y],Z1,Z2) :-
        insert(X,Z1,Z3,[]),
        insert_conditions(Y,Z3,Z2).

insert(..) :-                               % inserting one condition

% ==================
```

```
% GENERAL SUBROUTINES
% ====================

test(Goal) :-                                   % testing the evaluation time
                                                % of a goal
      time(U,M,S,Hs),
      test(Goal,U,M,S,Hs).
test(Goal,U,M,S,Hs) :-
      call(Goal),
      time(U1,M1,S1,Hs1),
      A is 3600 * M + S * 60 + Hs, B is 3600 * M1 + S1 * 60 + Hs1, D is
B - A,
      write('~J~M'),
      show(['Processing',time,D,'~J~M']).

show(..) :-                                     % writing a list of elements

count(..) :-                                    % counting the number of
                                                % elements in a list

is_a(..) :-                                     % testing whether a subgoal
                                                % is relational

incorrect(..) :-                                % testing for inexisting
                                                % relations and wrongly
                                                % formatted tuples

message(..) :-                                  % showing an error message

error(1,'relation does not exist').
error(2,'invalid number of attributes').
error(3,'tuple does not exist').

state :-                                        % showing information on
                                                % database contents
```

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS